System Calls

Learning Outcomes

- A high-level understanding of System Calls
  - Mostly from the user’s perspective
  - From textbook (section 1.6)
- Exposure architectural details of the MIPS R3000
  - Detailed understanding of the exception handling mechanism
  - From “Hardware Guide” on class web site
- Understanding of the existence of compiler function calling conventions
  - Including details of the MIPS 'C' compiler calling convention
- Understanding of how the application kernel boundary is crossed with system calls in general
  - Including an appreciation of the relationship between a case study (OS/161 system call handling) and the general case.

Operating System

System Calls

- Can be viewed as special procedure calls
  - Provides for a controlled entry into the kernel
  - While in kernel, they perform a privileged operation
  - Returns to original caller with the result
- The system call interface represents the abstract machine provided by the operating system.

A Brief Overview of Classes

- From the user’s perspective
  - Process Management
  - File I/O
  - Directories management
  - Some other selected Calls
  - There are many more
  - On Linux, see `man syscalls` for a list

Some System Calls For Process Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getpid()</td>
<td>Get thread id</td>
</tr>
<tr>
<td>gettid()</td>
<td>Get process id</td>
</tr>
<tr>
<td>waitpid</td>
<td>Wait for a child to terminate</td>
</tr>
<tr>
<td>write()</td>
<td>Write data to a file descriptor</td>
</tr>
<tr>
<td>execve()</td>
<td>Execute a new process</td>
</tr>
<tr>
<td>fork()</td>
<td>Create a child process</td>
</tr>
<tr>
<td>exit()</td>
<td>Exit the process</td>
</tr>
<tr>
<td>execve()</td>
<td>Execute a new process</td>
</tr>
<tr>
<td>fork()</td>
<td>Create a child process</td>
</tr>
<tr>
<td>exit()</td>
<td>Exit the process</td>
</tr>
<tr>
<td>execve()</td>
<td>Execute a new process</td>
</tr>
<tr>
<td>fork()</td>
<td>Create a child process</td>
</tr>
</tbody>
</table>
Some System Calls For File Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f = open(fd, how)</td>
<td>Open a file for reading, writing or both</td>
</tr>
<tr>
<td>s = dup3(fd)</td>
<td>Copy a file descriptor</td>
</tr>
<tr>
<td>m = read(buffer, size)</td>
<td>Read data from a file into a buffer</td>
</tr>
<tr>
<td>p = seek(buffer, offset, whence)</td>
<td>Move the file position</td>
</tr>
<tr>
<td>t = tell(buffer, offset)</td>
<td>Get the file position</td>
</tr>
</tbody>
</table>

Some System Calls For Directory Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m = mkdir(name)</td>
<td>Create an empty directory</td>
</tr>
<tr>
<td>s = truncate(name, named)</td>
<td>Create a new entry, named, pointing to named</td>
</tr>
<tr>
<td>t = truncate(name, size)</td>
<td>Create a new entry, size</td>
</tr>
<tr>
<td>u = umount(name, flag)</td>
<td>Unmount a file system</td>
</tr>
</tbody>
</table>

Some System Calls For Miscellaneous Tasks

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m = chdir(name)</td>
<td>Change the working directory</td>
</tr>
<tr>
<td>s = chdir(name, mode)</td>
<td>Change a file's protection is</td>
</tr>
<tr>
<td>t = settimeofday(ticks)</td>
<td>Set the clock to ticks</td>
</tr>
</tbody>
</table>

System Calls

- A stripped down shell:
  ```
  while (TRUE) {
    /* repeat forever */
    type_prompt(); /* display prompt */
    read_command(command, parameters); /* input from terminal */
    if (fork() != 0) { /* fork off child process */
      /* Parent code */
      waitpid(-1, &status, 0); /* wait for child to exit */
    } else {
      /* Child code */
      execve(command, parameters, 0); /* execute command */
    }
  }
  ```

The MIPS R2000/R3000

- Before looking at system call mechanics in some detail, we need a basic understanding of the MIPS R3000

Some Win32 API calls

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>closeHandle</td>
<td>Creates a new process</td>
</tr>
<tr>
<td>closeHandle</td>
<td>Creates a new process</td>
</tr>
<tr>
<td>dupHandle</td>
<td>Dupicate a file descriptor</td>
</tr>
<tr>
<td>read(buffer, size)</td>
<td>Read data from a file into a buffer</td>
</tr>
<tr>
<td>seek(buffer, offset, whence)</td>
<td>Move the file position</td>
</tr>
<tr>
<td>tell(buffer, offset)</td>
<td>Get the file position</td>
</tr>
<tr>
<td>GENERIC_READ</td>
<td>An unsecured permission</td>
</tr>
<tr>
<td>GENERIC_WRITE</td>
<td>An unsecured permission</td>
</tr>
<tr>
<td>GENERIC_EXECUTE</td>
<td>An unsecured permission</td>
</tr>
<tr>
<td>GENERIC_ALL</td>
<td>An unsecured permission</td>
</tr>
</tbody>
</table>

Some Win32 API calls

- Before looking at system call mechanics in some detail, we need a basic understanding of the MIPS R3000
MIPS R3000

- Load/store architecture
  - No instructions that operate on memory except load and store
  - Simple load/stores to/from memory from/to registers
    - Store word: `sw r4, (r5)`
    - Load word: `lw r3, (r7)`
    - Load contents of memory into r3 using address contained in r7
      - Delay of one instruction after load before data available in destination register
    - Must always an instruction between a load from memory and the subsequent use of the register.
- `lw, sw, lb, sb, lh, sh,...`

- Arithmetic and logical operations are register to register operations
  - E.g., `add r3, r2, r1`
  - No arithmetic operations on memory
- Example
  - `add r3, r2, r1 = r3 = r2 + r1`
- Some other instructions
  - `add, sub, and, or, xor, sll, srl`

MIPS R3000

- All instructions are encoded in 32-bit
- Some instructions have immediate operands
  - Immediate values are constants encoded in the instruction itself
  - Only 16-bit value
  - Examples
    - `add immediate: addi r2, r1, 2048`
    - `r2 = r1 + 2048`
    - `load immediate: li r2, 1234`
    - `r2 = 1234`

MIPS Registers

- User-mode accessible registers
  - 32 general purpose registers
  - `r0` hardwired to zero
  - `r31` the link register for jump and link (JAL) instruction
  - `hi/lo`
  - 2 * 32 bits for multiply and divide
  - `pc`
    - Not directly visible
    - Modified implicitly by jump and branch instructions

Branching and Jumping

- Branching and jumping have a branch delay slot
  - The instruction following a branch or jump is always executed prior to destination
- Examples
  - `li r2, 1`
  - `sw r0, (r3)`
  - `j 1f`
  - `li r2, 2`
  - `li r2, 3`
  - `l: sw r2, (r3)`

Jump and Link Instruction

- JAL is used to implement function calls
  - `r31 = PC+8`
- Return Address register (RA) is used to return from function call
  - `lw r4, (r6)`
  - `jr r31`
MIPS R3000

- RISC architecture – 5 stage pipeline

![MIPS 5-stage pipeline](image)

Coprocessor 0

- The processor control registers are located in CP0
  - Exception management registers
  - Translation management registers
- CP0 is manipulated using mtc0 (move to) and mfc0 (move from) instructions
  - mtc0/mfc0 are only accessible in kernel mode.

CP0 Registers

- Exception Management
  - c0_cause
    - Cause of the recent exception
  - c0_status
    - Current status of the CPU
  - c0_spn
    - Address of the instruction that caused the exception
  - c0_backaddr
    - Address accessed that caused the exception
- Miscellaneous
  - c0_prid
    - Processor Identifier
- Memory Management
  - c0_index
  - c0_random
  - c0_entryhi
  - c0_entrylo
  - c0_context
  - More about these later in course

![CP0 Registers](image)

c0_status

- For practical purposes, you can ignore most bits
  - Green background is the focus

- IM
  - Individual interrupt mask bits
  - 6 external
  - 2 software
- KU
  - 0 = kernel
  - 1 = user mode
- IE
  - 0 = all interrupts masked
  - 1 = interrupts enabled
  - Mask determined via IM bits
- c, p, o = current, previous, old

![c0_status](image)

![c0_cause](image)
### Exception Codes

<table>
<thead>
<tr>
<th>Exception Code</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Int</td>
<td>Interrupt</td>
</tr>
<tr>
<td>1</td>
<td>Mod</td>
<td>TLU modification</td>
</tr>
<tr>
<td>2</td>
<td>TLR</td>
<td>“T.L. load/T.L. store”</td>
</tr>
<tr>
<td>3</td>
<td>TLB</td>
<td>Address error on load/store or store respectively</td>
</tr>
<tr>
<td>4</td>
<td>MEL</td>
<td>Either an attempt to access outside kseg when in reset mode, or an attempt to read a word or half word at an unaligned address.</td>
</tr>
</tbody>
</table>

Table 3.2. Exception Code values: different kinds of exceptions

### Exception Vectors

<table>
<thead>
<tr>
<th>Program address</th>
<th>&quot;segment&quot;</th>
<th>Physical address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 0000</td>
<td>kseg0</td>
<td>0x0000 0000</td>
<td>TLB miss on kseg reference only</td>
</tr>
<tr>
<td>0x0000 0080</td>
<td>kseg0</td>
<td>0x0000 0080</td>
<td>All other exceptions</td>
</tr>
<tr>
<td>0x006c 0100</td>
<td>kseg1</td>
<td>0x006c 0100</td>
<td>Uncaught alternative kseg TLB miss entry point (used if 32-bit REN set)</td>
</tr>
<tr>
<td>0x006c 0100</td>
<td>kseg1</td>
<td>0x006c 0100</td>
<td>Uncaught alternative for all other exceptions, used if MSEL REN set</td>
</tr>
<tr>
<td>0x006c 0000</td>
<td>kseg1</td>
<td>0x006c 0000</td>
<td>The &quot;reset exception&quot;</td>
</tr>
</tbody>
</table>

Table 4.1. Reset and exception entry points (vectors) for R3000 family

### Simple Interrupt Walk-through

#### User Mode

- PC = 0x12345678
- Application

#### Kernel Mode

- PC = 0x12345678
- Application
- Interrupt

### Hardware exception handling

- Let's now walk through an exception
  - Assume an interrupt occurred as the previous instruction completed
  - Note: We are in user mode with interrupts enabled

- PC = 0x12345678
- EPC

- Cause
- Status
- KUo KUp Ke0 Ke1 KeK KeU KeC E0
Hardware exception handling

- Instruction address at which to restart after the interrupt is transferred to EPC

<table>
<thead>
<tr>
<th>PC</th>
<th>EPC</th>
<th>Cause</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12345678</td>
<td>0x12345678</td>
<td>?</td>
<td>KUo IEo KUp IEp KUc IEc</td>
</tr>
</tbody>
</table>

Interrupts disabled and previous state shifted along

<table>
<thead>
<tr>
<th>PC</th>
<th>EPC</th>
<th>Cause</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12345678</td>
<td>0x12345678</td>
<td>0</td>
<td>KUo IEo KUp IEp KUc IEc</td>
</tr>
</tbody>
</table>

Kernal Mode is set, and previous mode shifted along

<table>
<thead>
<tr>
<th>PC</th>
<th>EPC</th>
<th>Cause</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12345678</td>
<td>0x12345678</td>
<td>0</td>
<td>KUo IEo KUp IEp KUc IEc</td>
</tr>
</tbody>
</table>

Address of general exception vector placed in PC

<table>
<thead>
<tr>
<th>PC</th>
<th>EPC</th>
<th>Cause</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80000080</td>
<td>0x12345678</td>
<td>0</td>
<td>KUo IEo KUp IEp KUc IEc</td>
</tr>
</tbody>
</table>

Returning from an exception

- For now, lets ignore
  - how the exception is actually handled
  - how user-level registers are preserved
- Let’s simply look at how we return from the exception

<table>
<thead>
<tr>
<th>PC</th>
<th>EPC</th>
<th>Cause</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80000080</td>
<td>0x12345678</td>
<td>0</td>
<td>KUo IEo KUp IEp KUc IEc</td>
</tr>
</tbody>
</table>

CPU is now running in kernel mode at 0x80000080, with interrupts disabled

- All information required to
  - Find out what caused the exception
  - Restart after exception handling
    is in coprocessor registers

Badvaddr
Returning from an exception

PC: 0x8001234
EPC: 0x12345678

- This code to return is
  lw  r27, saved_epc
  nop
  jr  r27
  rfe

Load the contents of EPC which is usually moved earlier to somewhere in memory by the exception handler.

Returning from an exception

PC: 0x12345678
EPC: 0x12345678

- This code to return is
  lw  r27, saved_epc
  nop
  jr  r27
  rfe

Store the EPC back in the PC.

Returning from an exception

PC: 0x12345678
EPC: 0x12345678

- In the branch delay slot, execute a restore from exception instruction
  lw  r27, saved_epc
  nop
  jr  r27
  rfe

Returning from an exception

PC: 0x12345678
EPC: 0x12345678

- We are now back in the same state we were in when the exception happened
  lw  r27, saved_epc
  nop
  jr  r27
  rfe

Function Stack Frames

- Each function call allocates a new stack frame for local variables, the return address, previous frame pointer etc.
- Example: assume f1() calls f2(), which calls f3().

Function Stack Frames

- Each function call allocates a new stack frame for local variables, the return address, previous frame pointer etc.
- Example: assume f1() calls f2(), which calls f3().
### Function Stack Frames

- Each function call allocates a new stack frame for local variables, the return address, previous frame pointer etc.
- Example: assume f1() calls f2(), which calls f3().

![Function Stack Frames Diagram]

### Software Register Conventions

- Given 32 registers, which registers are used for:
  - Local variables?
  - Argument passing?
  - Function call results?
  - Stack Pointer?

<table>
<thead>
<tr>
<th>Reg No</th>
<th>Name</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-7</td>
<td>zero</td>
<td>Always return 0</td>
</tr>
<tr>
<td>8-11</td>
<td>a-e</td>
<td>preamble (context) preserved for use by assembler</td>
</tr>
<tr>
<td>12-15</td>
<td>v0-v3</td>
<td>Value interrupt F0 returned by subroutine</td>
</tr>
<tr>
<td>16-25</td>
<td>v4-v15</td>
<td>Argument First four parameters for a subroutine</td>
</tr>
<tr>
<td>26-31</td>
<td>v16-v31</td>
<td>Temporary, subroutines may use without saving</td>
</tr>
<tr>
<td>32</td>
<td>sp</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>33</td>
<td>a0-a31</td>
<td>Global pointer – Some immune systems maintain this to give copy access to &quot;static&quot; or &quot;external&quot; variables.</td>
</tr>
<tr>
<td>32</td>
<td>gp</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>33</td>
<td>tp</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>34</td>
<td>t0-t31</td>
<td>64 bit variable. Subroutines which need one can use this as a frame pointer</td>
</tr>
<tr>
<td>35</td>
<td>ra</td>
<td>Return address for subroutine</td>
</tr>
</tbody>
</table>

### Example Code

```c
void sixargs(int a, int b, int c, int d, int e, int f)
{
    return a + b + c + d + e + f;
}
```

```assembly
0040011c <main>:
    40011c: 27bdff08 addiu sp,sp,-40
    400120: afbf0024 sw ra,36(sp)
    400124: afbe0020 sw s8,32(sp)
    400128: 03a0f021 move s8,sp
    40012c: 24020005 li v0,5
    400130: afa20010 sw v0,16(sp)
    400134: 24020006 li v0,6
    400138: afca2014 sw v0,20(sp)
    40013c: 24040001 li a0,1
    400140: 24050002 li a1,2
    400144: 24060003 li a2,3
    400148: 0a10002c jal 4000b0 <sixargs>
    40014c: 24070004 li a3,4
    400150: afda0188 sw v0,24(sp)
    400154: 03e00008 move sp,sp
    400158: 8fbf0024 lw ra,36(sp)
    40015c: 8fbe0020 lw s8,32(sp)
    400160: 03e00008 jr ra
    400164: 27bf0028 addiu sp,sp,-40
```

### Stack Frame

- MIPS calling convention for gcc
  - Args 1-4 have space reserved for them
  - Local variables
  - Dynamic area
System Calls

Continued

User and Kernel Execution

• Simplistically, execution state consists of
  – Registers, processor mode, PC, SP
• User applications and the kernel have their
  own execution state.
• System call mechanism safely transfers
  from user execution to kernel execution
  and back.

System Call Mechanism in Principle

• Processor mode
  – Switched from user-mode to kernel-mode
    • Switched back when returning to user mode
• SP
  – User-level SP is saved and a kernel SP is initialised
    • User-level SP restored when returning to user-mode
• PC
  – User-level PC is saved and PC set to kernel entry
    point
    • User-level PC restored when returning to user-level
    – Kernel entry via the designated entry point must be
      strictly enforced

System Call Mechanism in Principle

• Registers
  – Set at user-level to indicate system call type and its
    arguments
    • A convention between applications and the kernel
    – Some registers are preserved at user-level or kernel-
      level in order to restart user-level execution
      • Depends on language calling convention etc.
    – Result of system call placed in registers when
      returning to user-level
      • Another convention
Why do we need system calls?

- Why not simply jump into the kernel via a function call????
  - Function calls do not
    - Change from user to kernel mode
      - and eventually back again
    - Restrict possible entry points to secure locations

Steps in Making a System Call

There are 11 steps in making the system call read (fd, buffer, nbytes)

MIPS System Calls

- System calls are invoked via a syscall instruction.
  - The syscall instruction causes an exception and transfers control to the general exception handler
  - A convention (an agreement between the kernel and applications) is required as to how user-level software indicates
    - Which system call is required
    - Where its arguments are
    - Where the result should go

OS/161 Systems Calls

- OS/161 uses the following conventions
  - Arguments are passed and returned via the normal C function calling convention
  - Additionally
    - Reg v0 contains the system call number
    - On return, reg a3 contains
      - 0: if success, v0 contains successful result
      - not 0: if failure, v0 has the errno.
      - v0 stored in errno
      - -1 returned in v0

OS/161 Systems Calls

- OS/161 uses the following conventions
  - Arguments are passed and returned via the normal C function calling convention
  - Additionally
    - Reg v0 contains the system call number
    - On return, reg a3 contains
      - 0: if success, v0 contains successful result
      - not 0: if failure, v0 has the errno.
      - v0 stored in errno
      - -1 returned in v0

CAUTION

- Seriously low-level code follows
- This code is not for the faint hearted

```
move $0, a3
addi $a1, $sp, 16
jal 0x1000 <read>
li $a2, 0x100
move $a0, $a2
blez $a0, $zero
```
User-Level System Call Walk Through – Calling read()

```c
int read(int filehandle, void *buffer, size_t size)
```

- Three arguments, one return value
- Code fragment calling the read function
- Args are loaded, return value is tested

Inside the read() syscall function part 1

```asm
0040068c <read>
```

- Appropriate registers are preserved
  - Arguments (a0-a3), return address (ra), etc.
- The syscall number (5) is loaded into v0
- Jump (not jump and link) to the common syscall routine

The read() syscall function part 2

```asm
00400640 <__syscall>
```

- Generate a syscall exception
  - Test success, if yes, branch to return from function
  - If failure, store code in errno
- Set read() result to -1
The read() syscall function part 2

Return to location after where read() was called

Summary

- From the caller’s perspective, the read() system call behaves like a normal function call
  - It preserves the calling convention of the language
- However, the actual function implements its own convention by agreement with the kernel
  - Our OS/161 example assumes the kernel preserves appropriate registers(s03, s8, sp, gp, ra).
- Most languages have similar support libraries that interface with the operating system.

System Calls - Kernel Side

- Things left to do
  - Change to kernel stack
  - Preserve registers by saving to memory (the stack)
  - Leave saved registers somewhere accessible to
    - Read arguments
    - Store return values
  - Do the “read()”
  - Restore registers
  - Switch back to user stack
  - Return to application

Note k0, k1 registers available for kernel use

exception:
move k1, sp /* Save previous stack pointer in k1 */
sfc0 k0, c0_status /* Get status register */
andi k0, k0, CST_Kup /* Check the we-were-in-user-mode bit */
beq k0, $0, 1f /* If clear, from kernel, already have stack */
nop /* delay slot */
/* Coming from user mode - load kernel stack into sp */
lw a0, curkstack /* load stack into sp */
lw sp, 0(k2) /* Use */
nop /* delay slot for the load */
1:
sfc0 k0, c0Cause /* Now, load the exception cause. */
) common_exception /* Kernel code */
nop

common_exception:
/*
* At this point:
*  - Interrupts are off. (The processor did this for us.)
*  - k0 contains the exception cause value.
*  - k1 contains the old stack pointer.
*  - sp points into the kernel stack.
*  - All other registers are untouched.
*/
/*
* Allocate stack space for 37 words to hold the trap frame,
* plus four more words for a minimal argument block.
*/
addi sp, sp, -164
These six stores are a "hack" to avoid confusing GDB.
You can ignore the details of why and how.

Save all the registers on the kernel stack.

Create a pointer to the base of the saved registers and state in the first argument register.

By creating a pointer to here of type struct trapframe *, we can access the user's saved registers as normal variables within 'C'.

The real work starts here.
We can now use the other registers (t0, t1) that we have preserved on the stack.

The real work starts here.
We can now use the other registers (t0, t1) that we have preserved on the stack.

/* Prepare to call mips_trap(struct trapframe *) */
addiu a0, sp, 16
jal mips_trap
/* call it */
 nop /* delay slot */

By creating a pointer to type struct trapframe *, we can access the user's saved registers as normal variables within 'C'.

/* The order here must match mips/include/trapframe .h. */
sw ra, 160(sp) /* dummy for gdb */
sw s8, 156(sp) /* save s8 */
sw sp, 152(sp) /* dummy for gdb */
sw sp, 148(sp) /* save sp */
sw k1, 144(sp) /* dummy for gdb */
sw k0, 140(sp) /* dummy for gdb */
sw sp, 144(sp) /* real saved sp */
nop /* delay slot for store */
mfc0 k1, c0_epc /* Copr.0 reg 13 == PC for exception */
sw k1, 152(sp) /* real saved PC */
Now we arrive in the ‘C’ kernel

/*
 * General trap (exception) handling function for mips.
 * This is called by the assembly-language exception handler once
 * the trapframe has been set up.
 */
void
mips_trap(struct trapframe *tf)
{
    u_int32_t code, isutlb, iskern;
    int savepl;

    /* The trap frame is supposed to be 37 registers long. */
    assert(sizeof(struct trapframe)==(37*4));

    /* Save the value of curspl, which belongs to the old context. */
    savepl = curspl;

    /* Right now, interrupts should be off. */
    curspl = SPL_HIGH;

    /* The kernel deals with whatever caused the exception
       - Syscall
       - Interrupt
       - Page fault
       - It potentially modifies the trapframe, etc
       - E.g., Store return code in v0, zero in a3
    */
    exception_return:
    /*     16(sp) no need to restore tf_vaddr */
    lw t0, 20(sp) /* load status register value into t0 */
    nop /* load delay slot */
    mtc0 t0, c0_status /* store it back to coprocessor 0 */
    /*     24(sp) no need to restore tf_cause */
    /* restore special registers */
    lw t1, 28(sp)
    lw t0, 32(sp)
    mthi t1
    mtlo t0
    /* load the general registers */
    lw ra, 36(sp)
    lw AT, 40(sp)
    lw v0, 44(sp)
    lw v1, 48(sp)
    lw a0, 52(sp)
    lw a1, 56(sp)
    lw a2, 60(sp)
    lw a3, 64(sp)
    lw gp, 68(sp) /* restore gp */
    /*     152(sp) stack pointer - below */
    lw sp, 96(sp) /* fetch saved sp (must be last) */
    jr k0 /* jump back */
    rfe /* in delay slot */

    .end common_exception

Note again that only k0, k1 have been trashed